Some cool and very useful library functions for C and C++ programs

Contents:

---

getopt

getopt is very useful for writing programs that take optional and/or required command line options. It can be used to parse command line arguments of the form:

```
    -o opt_arg  -o  ...
```

the man page for getopt has an example (man 3 getopt).

Example

An example program using getopt is here:

```
  cp -r ~newhall/public/getopts_example .
```

This is an example from one of my programs (it shows one way of handling required command line options):

```
// prints out error message when user tries to run with bad command line args or
// when user runs with the -h command line arg
void usage(void){
  fprintf(stderr,
      " usage:\n"
      "    ./server -p portnum [-h] [-c] [-f configfile] [-n secs]\n"
      "       -p  portnum:   use portnum as the listen port for server\n"
      "       -h:            print out this help message\n"
      "       -c:            run this server deamon in collector-only mode\n"
      "       -f conf_file:  run w/conf_file instead of /etc/server.config\n"
      "       -n secs:  how often damon sends its info to peers (default 5)\n"
      "\n");
}

// parse command line arguments
//   ac: argc value passed into main
//   av: argv value passed into main
// this function may set the value of global vars based on
// what command line options are present
//
void process_args(int ac, char *av[]){

  int c, p=0;  // p is a flag that we set if we get the -p command line option

  while(1){

    c = getopt(ac, av, ":p:chf:n:");   // "p:"  p option has an arg  "c"  does not
    if(c == -1) {
        break;   // all done parsing
    }

    switch(c){
```

```
      case 'h': usage(); exit(0); break;
      case 'p': port_num=optarg; p = 1; break;
      case 'c': collector_only = 1; break;
      case 'f': config_file=optarg; break;
      case 'n':
                sleep_secs=atoi(optarg);  // atoi converts a string to an int
                if(sleep_secs <= 0){        // (ex) atoi("1234") to int 1234
                  sleep_secs = 5;
                }
                break;
      case ':': fprintf(stderr, " -%c missing arg\n", optopt);
                usage(); exit(1); break;
      case '?': fprintf(stderr, " unknown arg %c\n", optopt);
                usage(); exit(1); break;
    } // switch
  } // while

  if(!p) {
    fprintf(stderr,"Error: server must be run with command line option -p\n");
    usage();
    exit(1);
  }
}

int main(int argc, char *argv[]) {

  process_args(argc, argv);
  ...
```

An example command line (assuming server is name of executable file):

```
$ ./server -p 1288 -c
```

readline

readline is a GNU library for reading in user input. It has support for all kinds of line editing capabilities that the user can use to edit the input line. For example, the user can move the curser to different positions in the line, and modify parts of the input string. See the readline man page and the readline homepage for complete information about the readline library: GNU readline homepage

One very nice feature of readline is that it mallocs up the space for the returned string. Thus, a program can easily support reading in any sized user input string by simply calling readline. Thus, even if you don't care about any of the line editing features of readline, it is still a handy way to read in user input.

To use readline, you need to include readline header files and explicitly link with the readline library:

```
$ gcc -o myprog  myprog.c -lreadline
```

Here is a very simple example program:

```
#include<stdlib.h>
#include<stdio.h>
#include<readline/readline.h>
#include<readline/history.h>

int main(){

   char* line;

   line = readline("enter a string:  ");  // readline allocates space for returned string
   if(line != NULL) {
```

```
            printf("You entered: %s\n", line);
            free(line);   // but you are responsible for freeing the space
        }
    }
```

As you run this, type in a line and before you hit ENTER try some of these commands to change the input string:

```
CNTRL-a   move curser to begining of input string
CNTRL-e   move curser to end of input string
CNTRL-b   move curser back one character
CNTRL-f   move curser forward one character
CNTRL-d   delete the character under the curser
CNTRL-k   kill the string from the curser to the end of the line
CNTRL-l   clear the screen and re-print the prompt and input string at the top
```

---

ascii escape codes

You can use ascii escape codes in C strings to print text to the terminal in different colors (red, blue, ...), with diffent attributes (bold, underscore, blink, ...) The general form of an escape sequence is the following (any missing component is assumed to be zero):

```
\e[attribute code;text color code;background color codem
```

The values for these are:

```
Attribute codes:
----------------
0=none 1=bold 4=underscore 5=blink 7=reverse 8=concealed
Text color codes:
-----------------
30=black 31=red 32=green 33=yellow 34=blue 35=magenta 36=cyan 37=white
Background color codes:
----------------------
40=black 41=red 42=green 43=yellow 44=blue 45=magenta 46=cyan 47=white
```

The effect is ended by specifying:

```
\e[0m
```

For example, to print out the string Hello in red:

```
printf("\e[0;31mHello\e[0m");
```

For example, to print out the string Hello in bold blue:

```
printf("\e[1;34mHello\e[0m");
```

---

ncurses

ncurses is a library for terminal-independent I/O to character screens. It can be used to create character-based user interfaces to terminal windows.

See the man page for ncurses for more information. Also, here is an ncurses HOWTO

To use ncurses library in your program, you need to include the ncurses.h header file, and link in the ncurses library:

```
gcc -o myprog myprog.c -lncurses
```

Here is a very simple example of printing to the terminal using different colors:

```
#include <ncurses.h>

int main(){

  initscr();  // initialize the ternimal in curses mode

  start_color(); // start color mode

  init_pair(1, COLOR_RED, COLOR_BLUE); // define a forground, background pair
  attron(COLOR_PAIR(1));   // enable for/back ground color to use
  printw("Hello World\n"); // print string to curses window
  attroff(COLOR_PAIR(1));
  refresh();               // forces printw output to curses window
  getch();  // just wait for user input

  init_pair(2, COLOR_YELLOW, COLOR_MAGENTA); // another for/background pair
  attron(COLOR_PAIR(2));   // enable for/back ground color to pair # 2
  printw("Hello World\n"); // print string to curses window
  attroff(COLOR_PAIR(2));
  refresh();               // forces printw output to curses window
  getch();  // just wait for user input

  printw("Hello World\n");  // print using default for/backround colors
  refresh();               // forces printw output to curses window
  getch();  // just wait for user input

  endwin();  // end curses mode
}
```

Here are some reference:
programing with curses: Unix C library for Screen Manipulation O'Reilly publication.
ncurses HOWTO The Linux Documentation Project